

图文 065、案例实战：百万级数据误处理导致的频繁Full GC问题优化

917 人次阅读 2019-09-03 07:00:00

详情 评论



狸猫技术窝

进店逛

案例实战： 百万级数据误处理导致的频繁Full GC问题优化

狸猫技术窝专栏上新，基于**真实订单系统**的消息中间件（mq）实战，重磅推荐：



相关频道



从 0 开始
战高手
已更新1

未来3个月，我的好朋友原子弹大侠将带你一起，全程实战，360度死磕MQ

(点击下方蓝字进行试听)

[从 0 开始带你成为消息中间件实战高手](#)

重要说明：

如何提问：每篇文章都有评论区，大家可以尽情在评论区留言提问，我都会逐一答疑

(ps：评论区还精选了一些小伙伴对**专栏每日思考题**的作答，有的答案真的非常好！大家可以通过看别人的思路，启发一下自己，从而加深理解)

如何加群：购买了狸猫技术窝专栏的小伙伴都可以加入**狸猫技术交流群**。

(群里有不少**一二线互联网大厂**的助教，大家可以一起讨论交流各种技术)

具体**加群方式**请参见文末。

(注：以前通过其他专栏加过群的同学就不要重复加了)

1、事故场景

有一次一个线上系统进行了一次版本升级，结果升级过后才半小时，突然之间收到运营和客服雪花般的反馈，说这个系统对应的前端页面无法访问了，所有用户全部看到的是一片空白和错误信息。

这个时候通过监控报警平台也收到雪花般的报警，发现线上系统所在机器的CPU负载非常高，持续走高，甚至直接导致机器都宕机了。所以系统对应的前端页面当然是什么都看不到了。

2、CPU负载高原因分析

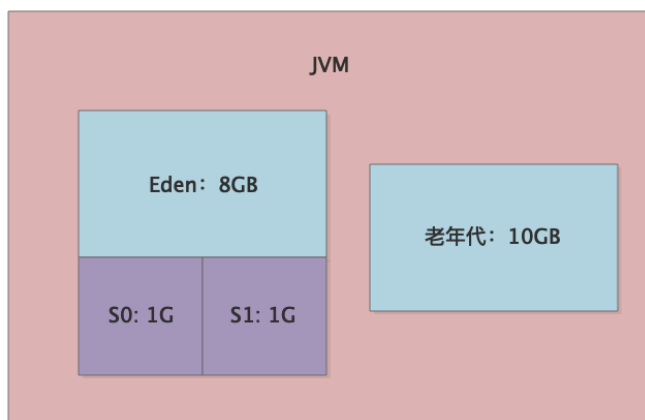
上篇文章已经给大家总结过CPU负载高的原因了，这里我们直接说结论，看了一下监控和jstat，发现Full GC非常频繁，基本上两分钟就会执行一次Full GC，而且每次Full GC耗时非常长，在10秒左右！

所以直接入手尝试进行Full GC的原因定位。

3、Full GC频繁的原因分析

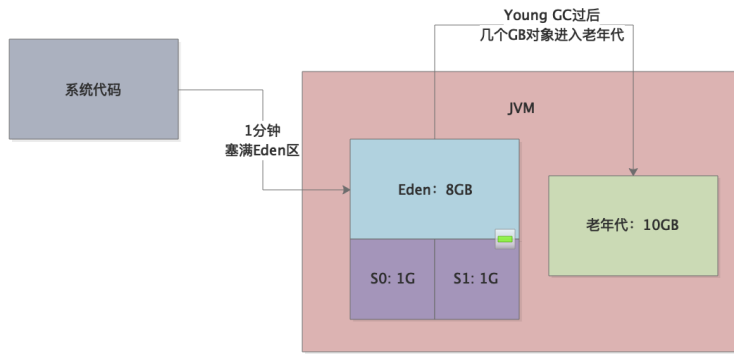
上篇文章也给大家总结过Full GC频繁的几种常见原因了，其实分析Full GC频繁的原因，最好的工具不是监控平台，就是最实用的jstat工具，直接看线上系统运行时候的动态内存变化模型，什么问题都立马出来了。

基于jstat一分析发现了很大的问题，当时这个系统因为主要是用来进行大量数据处理然后提供数据给用户查看的，所以当时可是给JVM的堆分配了20G的内存，其中10G给了年轻代，10G给了老年代，如下图所示：



这么大的年轻代，结果大家能猜到jstat看到什么现象吗？Eden区大概1分钟左右就会塞满，然后就会触发一次Young GC，而且Young GC过后有几个GB的对象都是存活的会进入老年代！

如下图所示。



这说明什么？这说明系统代码运行的时候在产生大量的对象，而且处理的极其的慢，经常在1分钟过后Young GC以后还有很多对象在存活，才会导致大量对象进入老年代中！这真是让人非常的无奈。

所以就是因为这个内存运行模型，才导致了平均两分钟就会触发一次Full GC，因为老年代两分钟就塞满了，而且老年代因为内存量很大，所以导致一次Full GC就要10秒的时间！

大家想象一下，系统每隔2分钟就要暂停10秒，对用户是什么感觉！

更有甚者，普通的4核机器根本撑不住这么频繁，而且这么耗时的Full GC，所以这种长时间Full GC直接导致机器的CPU频繁打满，负载过高，也导致了用户经常无法访问系统，页面都是空白的！

4、以前那套GC优化策略还能奏效吗？

那么大家想一下，以前我们给大家说过的那套GC优化策略此时还能奏效吗？

即把年轻代调大一些，给Survivor更多的内存空间，避免对象进入老年代。

明显不行，这个运行内存模型告诉我们，即使你给年轻代更大空间，甚至让每块Survivor区域达到2GB或者3GB，但是一次Young GC过后，还是会因为系统处理过慢，导致几个GB的对象存活下来，你Survivor区域死活都是放不下的！

所以这个时候就不是简单优化一下JVM参数就可以搞定的。

这个系统明显是因为代码层面有一定的改动和升级，直接导致了系统加载过多数据到内存中，而且对过多数据处理的还特别慢，在内存里几个GB的数据甚至要处理一分多钟，才能处理完毕。

这是明显的代码层面的问题了，其实要优化这个事故，就必须得优化代码，而不是简单的JVM参数！

我们需要避免代码层面加载过多的数据到内存里去处理，这是最核心的一个点。

5、复杂的业务逻辑，自己都看不懂了怎么办？

说优化代码，说起来很简单，但是实际做起来呢？

有很多系统的代码都特别的复杂，别说别人看不懂了，自己可能写的代码过了几个月自己都看不懂了！所以直接通过走读代码来分析问题所在是很慢的！

我们必须有一个办法可以立马就定位到系统里到底是什么样的对象太多了占用了过多的内存，这个时候就得使用一个工具了：**MAT**

这个工具我们上篇文章介绍过，今天给大家深入讲讲他的使用，给出一些界面的截图出来。

6、准备一段示范用的代码

现在我们来准备一段示范用的代码，在代码里创建一大堆的对象出来，然后我们尝试获取他的dump内存快照，再用MAT来进行分析。

```
public class Demo1 {
    public static void main(String[] args) throws Exception {
        List<Data> datas = new ArrayList<Data>();
        for(int i = 0; i < 10000; i++) {
            datas.add(new Data());
        }
        Thread.sleep(1 * 60 * 60 * 1000);
    }
    static class Data {
    }
}
```

这段代码大家可以看到，非常的简单，就是在代码里创建10000个自定义的对象，然后就陷入一个阻塞状态就可以了，大家可以把这段代码运行起来。

7、获取jvm进程的dump快照文件

先在本地命令行运行一下jps命令，查看一下启动的jvm进程的PID，如下所示：

```
1190 Jps
1176 Launcher
1177 Demo1
1151
```

明显能看到，我们的Demo1这个类的进程ID为1177

接着执行下面的jmap命令可以导出dump内存快照：

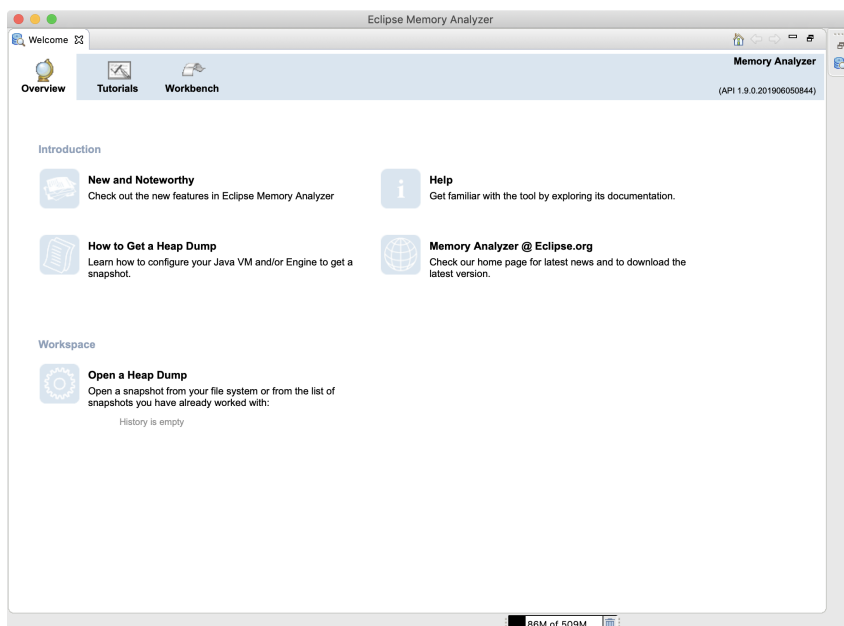
```
jmap -dump:live,format=b,file=dump.hprof 1177
```

8、使用MAT分析内存快照

大家在下一个步骤，务必要注意到，如果是线上导出来的dump内存快照，很多时候可能都是几个GB的

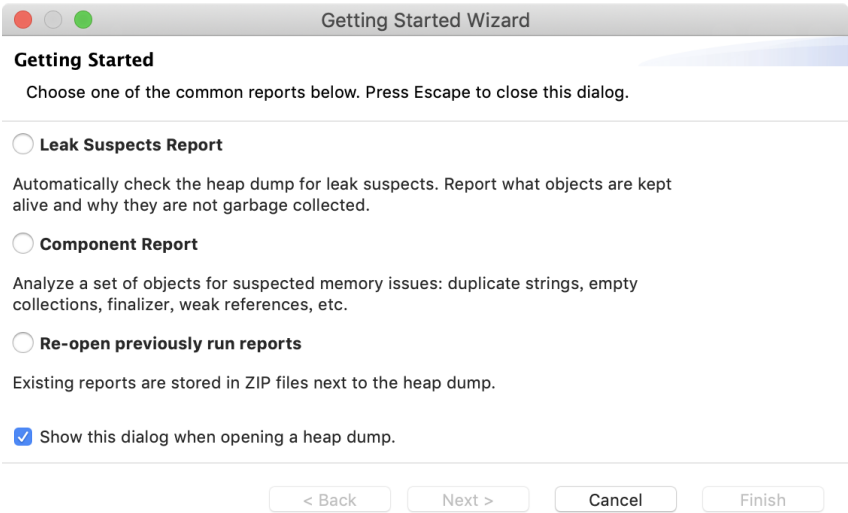
比如我们这里就是8个多GB的内存快照，所以就务必按照上节课所说的，在MAT的MemoryAnalyzer.ini文件里，修改他的启动堆大小为8GB。

接着就是打开MAT软件，如下图所示：

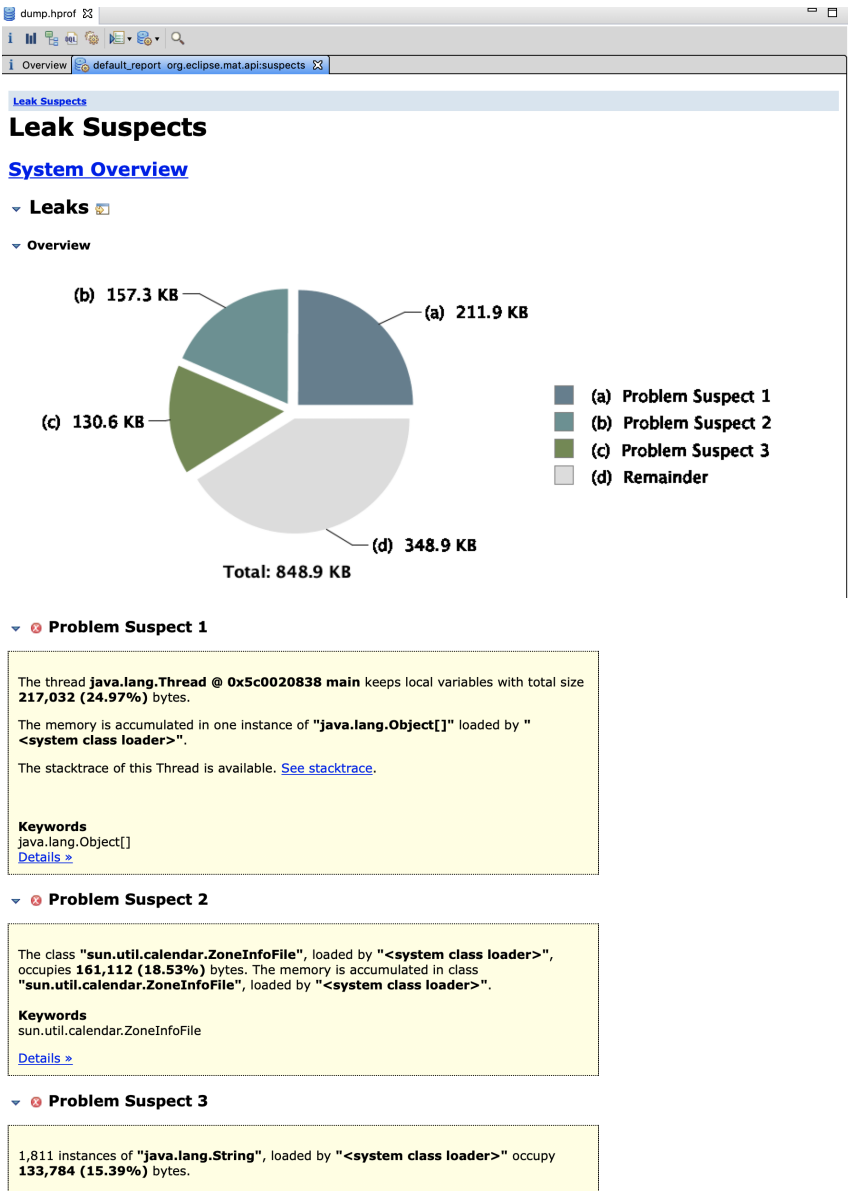


然后选择其中的“Open a Heap Dump”打开自己的那个dump快照即可

接着会看到下图的样子，大家如果跟我一样用的是最新版本的MAT，那么打开dump快照的时候就会提示你，要不要进行内存泄漏的分析，就是Leak Suspects Report，你一般勾选是即可。



接着大家会看到如下的图：



从上面图里可以很清晰的看到了，别人都告诉你了，可能存在的内存泄漏的问题，尤其是第一个问题，“Problem Suspect 1”，他的英文里很清晰的告诉你了，`java.lang.Thread main`，就是说main线程，通过局部变量引用了占据24.97%内存的对象。

而且他告诉你那是一个java.lang.Object[]数组，这个数组占据了大量的内存。

那么大家肯定想要知道，到底这个数组里是什么东西呢？

大家可以看到上面的“Problem Suspect 1”框框的里面最后一行是一个超链接的“Details”，大家点击进去就可以看到详细的说明了。

▼ Shortest Paths To the Accumulation Point

Class Name	Shallow Heap	Retained Heap
java.lang.Object[14053] @ 0x5c0048418	56,232	216,232
elementData java.util.ArrayList @ 0x5c00206a8	24	216,256
<Java Local> java.lang.Thread @ 0x5c0020838_main Thread	120	217,032

▼ Accumulated Objects in Dominator Tree

Class Name	Shallow Heap	Retained Heap	Percentage
java.lang.Thread @ 0x5c0020838_main	120	217,032	24.97%
java.util.ArrayList @ 0x5c00206a8	24	216,256	24.88%
java.lang.Object[14053] @ 0x5c0048418	56,232	216,232	24.88%
Demo1\$Data @ 0x5c000bea0	16	16	0.00%
Demo1\$Data @ 0x5c000cbe0	16	16	0.00%
Demo1\$Data @ 0x5c000cbf0	16	16	0.00%
Demo1\$Data @ 0x5c000cc00	16	16	0.00%
Demo1\$Data @ 0x5c000cc10	16	16	0.00%
Demo1\$Data @ 0x5c000cc20	16	16	0.00%
Demo1\$Data @ 0x5c000cc30	16	16	0.00%
	16	16	0.00%

通过这个详细的说明，尤其是那个“Accumulated Objects in Dominator Tree”，在里面我们可以看到，java.lang.Thread main线程中引用了一个java.util.ArrayList，这里面是一个java.lang.Object[]数组，数组里的每个元素都是Demo1\$Data对象实例。

到此为止，就很清楚了，到底是什么对象在内存里占用了过大的内存。所以大家通过这个小小的范例就可以知道，你的系统中那些超大对象到底是什么，用MAT分析内存是非常方便的。

9、追踪线程执行堆栈，找到问题代码

一旦发现某个线程在执行过程中创建了大量的对象之后，就可以尝试找找这个线程到底执行了哪些代码才创建了这些对象

如下图所示，先点击页面中的一个“See stacktrace”，可然后就会进入一个线程执行代码堆栈的调用链：

▼ Description

The thread **java.lang.Thread @ 0x5c0020838 main** keeps local variables with total size **217,032 (24.97%)** bytes.



The memory is accumulated in one instance of "**java.lang.Object[]**" loaded by "**<system class loader>**".

The stacktrace of this Thread is available. [See stacktrace.](#)

Keywords

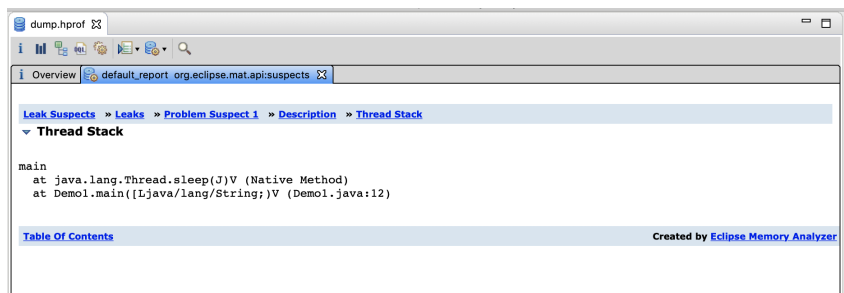
java.lang.Object[]

▼ Shortest Paths To the Accumulation Point 🗺️

Class Name	Shallow Heap	Retained Heap
 java.lang.Object[14053] @ 0x5c0048418	56,232	216,232
 elementData java.util.ArrayList @ 0x5c00206a8	24	216,256
 <Java Local> java.lang.Thread @ 0x5c0020838 main Thread	120	217,032

▼ Accumulated Objects in Dominator Tree 🗺️

Class Name	Shallow Heap	Retained Heap	Percentage
 java.lang.Thread @ 0x5c0020838 main	120	217,032	24.97%
 java.util.ArrayList @ 0x5c00206a8	24	216,256	24.88%



在当时而言，我们就是按照这个方法追踪到了线上系统某个线程的执行堆栈，最终发现的是这个线程执行“String.split()”方法导致产生了大量的对象

那么到底是为什么呢？接着我们来分析一下“String.split()”这个方法。

10、为什么“String.split()”会造成内存泄漏？

其实原因很简单，当时这个线上系统用的是JDK 1.7。

在JDK 1.6的时候，String.split()方法的实现是这样子的，比如你有一个字符串，“Hello World Ha Ha”，然后你按照空格来切割这个字符串，应该会出来四个字符串“Hello” “World” “Ha” “Ha”，大家应该理解这个吧？

那么JDK 1.6的时候，比如“Hello World Ha Ha”这个原始字符串底层是基于一个数组来存放那些字符的

比如[H,e,l,l,o,,W,o,r,l,d,,H,a,,H,a]这样的数组。然后切割出来的“Hello”字符串他不会对应一个新的数组，而是直接映射到原来那个字符串的数组，采用偏移量表明自己是对应原始数组中的哪些元素。

比如说“Hello”字符串可能就是对应[H,e,l,l,o,,W,o,r,l,d,,H,a,,H,a]数组中的0~4位置的元素。

但是JDK 1.7的时候，他的实现是给每个切分出来的字符串都创建一个新的数组，比如“Hello”字符串就对应一个全新的数组，[H,e,l,l,o]。

所以当时那个线上系统的处理逻辑，就是加载大量的数据出来，可能有的时候一次性加载几十万条数据，数据主要是字符串

然后对这些字符串进行切割，每个字符串都会切割为N个小字符串。

这就瞬间导致字符串数量暴增几倍甚至几十倍，这就是系统为什么会频繁产生大量对象的根本原因!!!

因为在本次系统升级之前，是没有String.split()这行代码的，所以当时系统基本运行还算正常，其实一次加载几十万条数据量也很大，当时基本上每小时都会有几次Full GC，不过基本都还算正常。

只不过系统升级之后代码加入了String.split()操作，瞬间导致内存使用量暴增N倍，引发了上面说的每分钟一次Young GC，两分钟一次Full GC，根本原因就在于这行代码的引入。

11、代码如何进行优化？

后来紧急对这段代码逻辑进行了优化，避免对几十万条数据每一条都执行String.split()这行代码让内存使用量暴增N倍，然后再对那暴增N倍的字符串进行处理。

就当时而言，其实String.split()这个代码逻辑是可用可不用的，所以直接去除就行了。

但是如果从根本而言，说白了，还是这种大数据量处理的系统，一次性加载过多数据到内存里来了，所以后续对代码还是做了很多的优化

比较核心的思路，就是开启多线程并发处理大量的数据，尽量提升数据处理完毕的速度，这样到触发Young GC的时候避免过多的对象存活下来。

12、今日思考题

今天给大家留一个小作业，可以去自己线上系统dump一个内存快照出来，用MAT练习一下分析自己系统的内存

看看都有哪些较大的对象，什么线程创建的，看看线程执行堆栈，体验一下分析内存的感觉。

End

专栏版权归公众号**狸猫技术窝**所有

未经许可不得传播，如有侵权将追究法律责任

如何加群？

添加微信号：Lvgu0715_（微信名：绿小九），狸猫技术窝管理员

发送 Jvm专栏的购买截图

由于是人工操作，发送截图后请耐心等待被拉群

最后再次提醒：通过其他专栏加过群的同学，就不要重复加了

狸猫技术窝其他精品专栏推荐：

[21天互联网java进阶面试训练营（分布式篇）](#)